Chapter 5 System Modelling

5.1 Context models

5.2 Interactions model

5.3 Structural models

5.4 Behavioral models

5.5 Model-driven architecture

Five UML Diagram Types

1. **Activity Diagrams**:
   - Depict the activities involved in a process or in data processing.
2. **Use Case Diagrams**:
   - Illustrate the interactions between a system and its environment.
3. **Sequence Diagrams**:
   - Demonstrate interactions between actors and the system as well as between system components.
4. **Class Diagrams**:
   - Display the object classes in the system and the associations between these classes.
5. **State Diagrams**:
   - Reveal how the system reacts to internal and external events.

Each type of UML diagram serves a specific purpose and offers different insights into the system's design, functionality, or interaction.

5.1 Context models

Context Models in System Specification

1. **Defining System Boundaries**:
   - Early in the specification process, decide what is and isn't part of the system.
   - Work with stakeholders to determine what functionalities should be included.

    - Make decisions early to limit costs and time.
2. **System vs. Environment Boundaries**:
    - Sometimes the boundary is clear, especially if replacing an existing system.
    - Flexibility exists in determining what constitutes the boundary.
3. **Example: Mentcare System**:
    - Must decide if the system will only collect consultation info or also collect personal patient info.
    - Consider advantages and disadvantages, such as data duplication and system speed.
4. **User Base Diversity**:
    - Diverse user bases may require configurable systems that can adapt to varying needs.
5. **Non-Technical Factors**:
    - Boundaries may be influenced by social and organizational concerns, such as avoiding a difficult manager or increasing system costs for departmental expansion.
6. **Context Definition and Dependencies**:
    - After deciding on boundaries, define the system's context and its dependencies on its environment.
    - Architectural models often serve as the first step.
7. **Relation with Other Systems**:
    - Context models show that the environment often includes several other automated systems but do not detail the types of relationships.
    - Other models, like business process models, can be used in conjunction.
8. **UML Activity Diagrams**:
    - Can be used to show business processes where systems are used.
    - Detail the flow of control from one activity to another.
9. **Critical Functions Example**:
    - The Mentcare system must implement legal safeguards for involuntary detention of patients.
10. **Flow and Coordination in Activity Diagrams**:
    - Arrows indicate the flow of work; solid bars indicate activity coordination.
    - Activities can be concurrent or sequential based on the diagram.

This summary captures the intricacies of defining system boundaries, making considerations for diverse user bases, and employing various models, like UML activity diagrams, to better understand the system's context and dependencies.

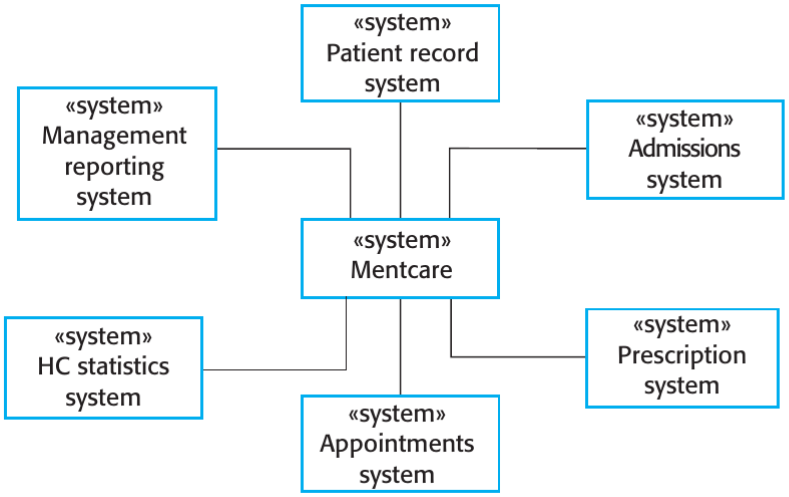

Figure 5.1 The context of the Mentcare system

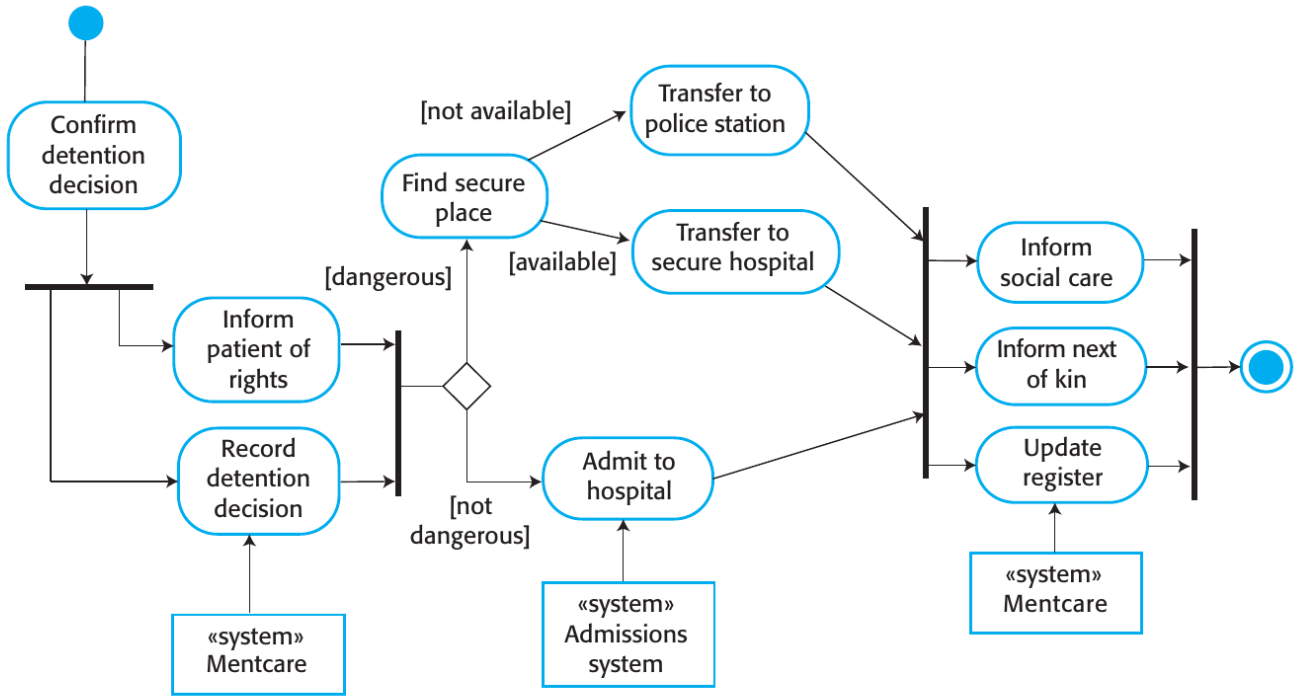

Figure 5.2 A process model of involuntary detention

these relations may affect the requirements and design of the system being defined and so must be taken into account. Therefore, simple context models are used along

5.2 Interactions model

Interaction Models in Software Systems

1. **Types of Interactions**:
   - User interaction (user inputs and outputs)
   - System-to-system interaction
   - Component-to-component interaction within a software system
2. **Importance of Interaction Modeling**:
   - Helps identify user requirements

- Highlights potential communication problems
- Aids in understanding system performance and dependability

3. **Approaches to Interaction Modeling**:
   i. **Use Case Modeling**:
      - Primarily used for interactions between a system and external agents (either human users or other systems)
   ii. **Sequence Diagrams**:
      - Used to model interactions between system components
      - External agents may also be included
4. **Complementary Use**:
   - Use case models and sequence diagrams can be used together to present interactions at different levels of detail
   - High-level use case interactions can be further detailed using sequence diagrams
5. **UML Communication Diagrams**:
   - Another tool for modeling interactions
   - Not discussed in detail as they are considered an alternative representation of sequence diagrams

These key points outline the types of interactions that can occur in a system, the importance of modeling these interactions, and the methodologies used for this purpose.

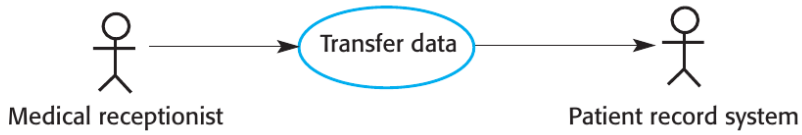

Figure 5.3 Transfer-data use case

5.2.1 use case modelling

1. **Origins and Utility**:
   - Developed by Ivar Jacobsen in the 1990s.
   - Part of the Unified Modeling Language (UML).
   - More useful in early stages of system design than in requirements engineering.
2. **Basic Components**:
   i. **Use Case**: Represents a discrete task involving external interaction with a system.
   ii. **Actors**: External entities interacting with the use case, represented as stick figures.
   iii. **Ellipse**: A simple graphical representation of a use case.
3. **Actors and Directionality**:
   - Actors can be human or other systems.
   - UML formally uses lines without arrows to show interactions.
   - Arrows can be used informally to indicate the initiator of the transaction.
4. **Detailing a Use Case**:
   - Use case diagrams provide a simple overview.
   - Additional detail can be added through:
     a. Textual description
     b. Structured description in a table
     c. Sequence diagrams

- The format depends on the level of detail required.

5. **Composite Use Case Diagrams**:
   - Show multiple use cases together.
   - Can be categorized by actor or functionality.
   - May require several diagrams for complex systems.
6. **Advanced UML Constructs**:
   - UML allows for sharing parts of a use case in other diagrams.
   - Often found to be confusing by end-users, so not universally recommended.

The summary outlines the components, utility, and methods of detailing use case models as part of the Unified Modeling Language (UML). It also touches upon the intricacies and limitations of using advanced UML constructs for use cases.

Mentcare system: Transfer data	
Actors	Medical receptionist, Patient records system (PRS)
Description	A receptionist may transfer data from the Mentcare system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

Figure 5.4 Tabular description of the Transfer-data use case

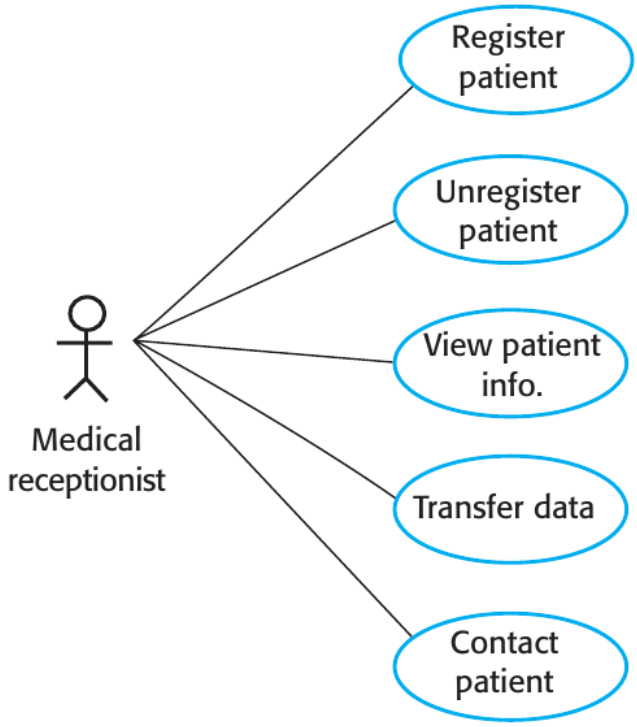

Figure 5.5 Use cases involving the role "Medical receptionist"

5.2.1 Sequence Diagrams

1. **Purpose and Scope**:
   - Used to model interactions between actors and objects, and among objects themselves.
   - Rich syntax but focus here is on basics.
2. **Core Elements**:
   i. **Objects and Actors**: Listed along the top of the diagram.
   ii. **Dotted Line**: Represents the lifeline of an object or actor, indicating involvement over time.
   iii. **Annotated Arrows**: Show interactions between objects and actors.
   iv. **Rectangle on Dotted Lines**: Indicates the lifeline or time an object instance is involved.
3. **Reading Sequence**:
   - Interactions are read from top to bottom.
   - Annotations on arrows indicate calls, parameters, and return values.
4. **Alternatives and Conditions**:
   - "Alt" box used to indicate alternative interactions.
   - Conditions for alternatives are shown in square brackets.
5. **Example Scenarios**:
   i. **View Patient Information**:
      - Triggered by a medical receptionist.
      - Involves security checks and conditional display of information.
   ii. **Transfer to Patient Record System (PRS)**:
      - Receptionist logs onto PRS.
      - Two options for data transfer, both requiring authorization checks.

- PRS issues a status message upon completion.

6. **Level of Detail**:
   - Not necessary to include every interaction, especially for early-stage modeling.
   - Some interactions might depend on later implementation decisions.

Sequence diagrams in UML serve as a valuable tool for modeling interactions within a system. They feature a range of elements that detail object lifetimes, interactions, and conditions for alternative pathways. The level of detail can be adjusted according to the stage of development, making these diagrams versatile for different uses.

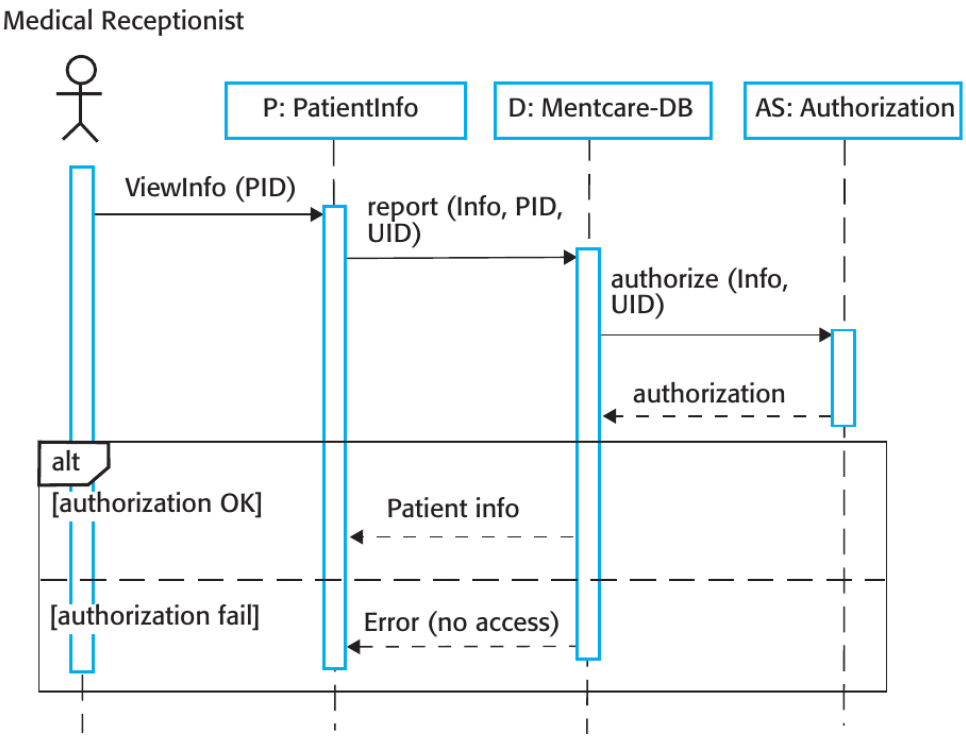

Figure 5.6 Sequence diagram for View patient information

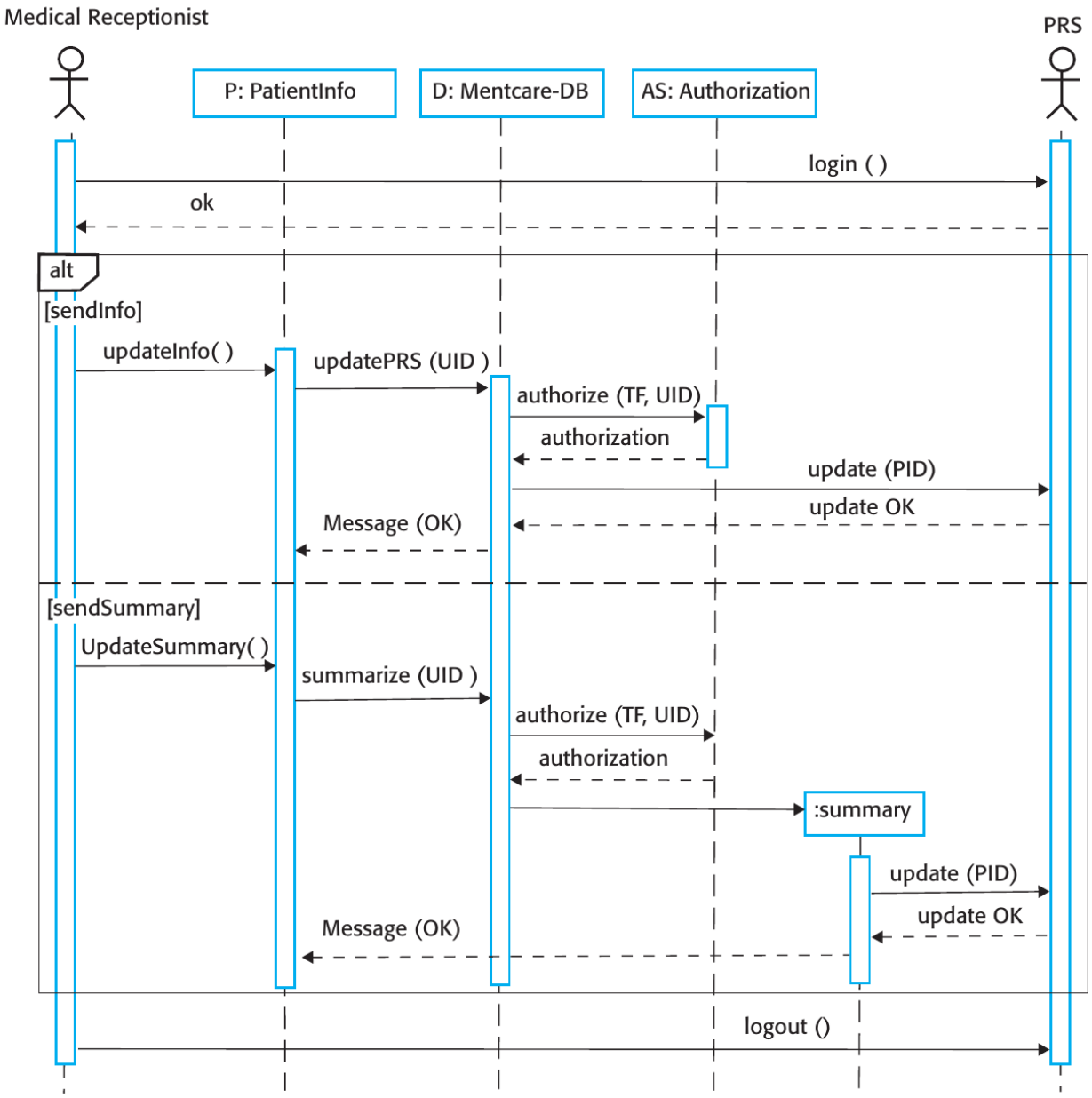

Figure 5.7 Sequence diagram for Transfer Data

to be uploaded to a national PRS (patient records system). You can read this diagram as follows:

5.3 Structural models

5.3.1 Class Diagrams

1. **Purpose and Utility**:
    - Used in object-oriented system modeling to depict classes and their associations.
    - Early stages focus on real-world objects like patients, prescriptions, etc.
    - Can be used at different levels of detail.
2. **Basic Elements**:
    i. **Object Class**: Represents a kind of system object. Shown as a box with the class name.

ii. **Association**: Links between classes, indicating a relationship. Represented by a line between class boxes.

3. **Levels of Detail**:
   - Initial modeling identifies essential real-world objects as classes.
   - Simple class diagrams may not specify the nature of associations between classes.
   - More detailed diagrams can name the associations.
4. **Multiplicity in Associations**:
   - Indicates how many objects are involved in an association.
   - Examples: 1:1, 1:*, 1..4, etc.
5. **Similarity to Semantic Data Models**:
   - Resemble semantic data models used in database design.
   - Entities in a semantic data model can be thought of as simplified object classes.
6. **Attributes and Operations**:
   - To add more detail, include attributes (characteristics) and operations (functions) of the classes.
   - Shown by extending the rectangle that represents a class.
     a. Class name in the top section.
     b. Attributes in the middle section.
     c. Operations in the lower section.

Class diagrams in UML serve as a foundational tool in object-oriented system modeling, allowing for different levels of detail depending on the development stage. They can show simple to complex relationships between classes, include multiplicities to indicate the number of objects in an association, and can be extended to include attributes and operations for each class.

Figure 5.8 UML Classes and association

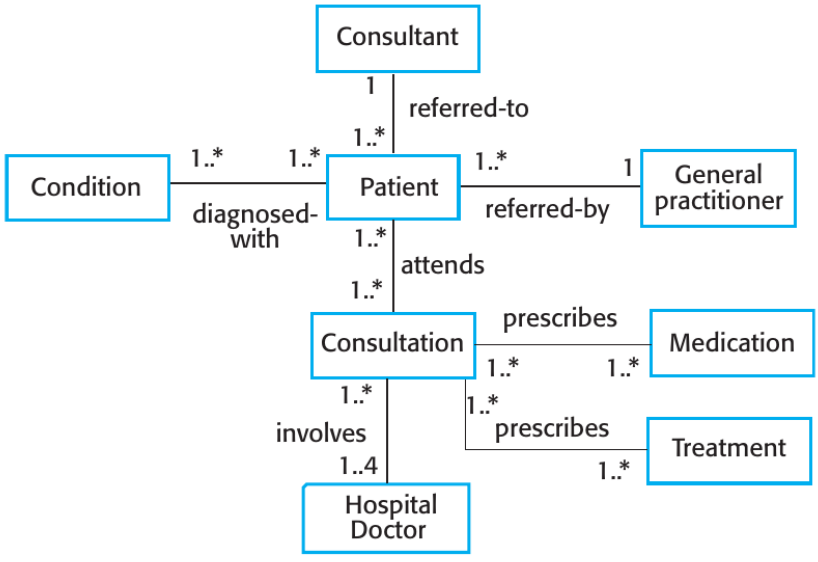

Figure 5.9 Classes and associations in the Mentcare system

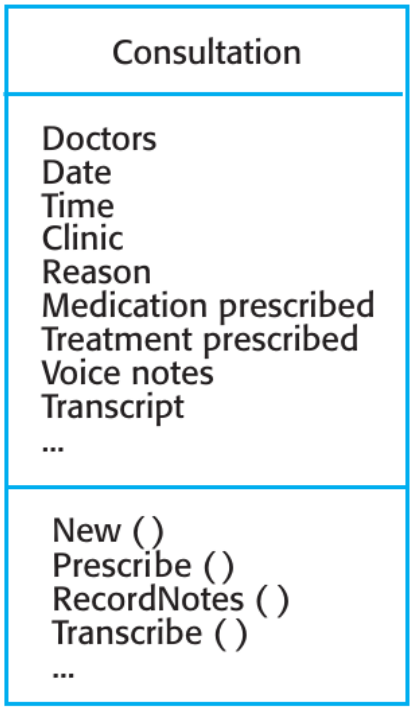

Figure 5.10 A Consultation class

5.3.2 generalization

1. **Concept of Generalization**:
   - A technique to manage complexity by focusing on general classes and their characteristics.
   - Example: Squirrels and rats are generalized as "rodents," sharing common characteristics like teeth for gnawing.
2. **Importance in System Design**:
   - Allows for centralized maintenance of common information.
   - Facilitates easier updates and changes, as you only need to modify the general class.
3. **Implementation in Object-Oriented Languages**:
   - In languages like Java, generalization is implemented through class inheritance mechanisms.
4. **UML Representation**:
   - UML uses a specific type of association to denote generalization.
   - Represented by an arrowhead pointing to the more general class.
5. **Inheritance and Attributes**:
   - Subclasses inherit attributes and operations from their superclasses.
   - Subclasses can add more specific attributes and operations.
6. **Examples**:
   - General practitioners and hospital doctors can be generalized as doctors.
   - Hospital doctors may have attributes like staff number and pager, which general practitioners do not.
   - Lower-level classes like "Trainee Doctor," "Registered Doctor," and "Consultant" fall under "Hospital Doctor" and add their own specific attributes and operations.

Generalization is a key concept in both everyday reasoning and system modeling, helping to manage complexity by focusing on shared characteristics. In system design, it simplifies maintenance and updates. It is

implemented in object-oriented languages through inheritance, and its relationships are explicitly depicted in UML diagrams.

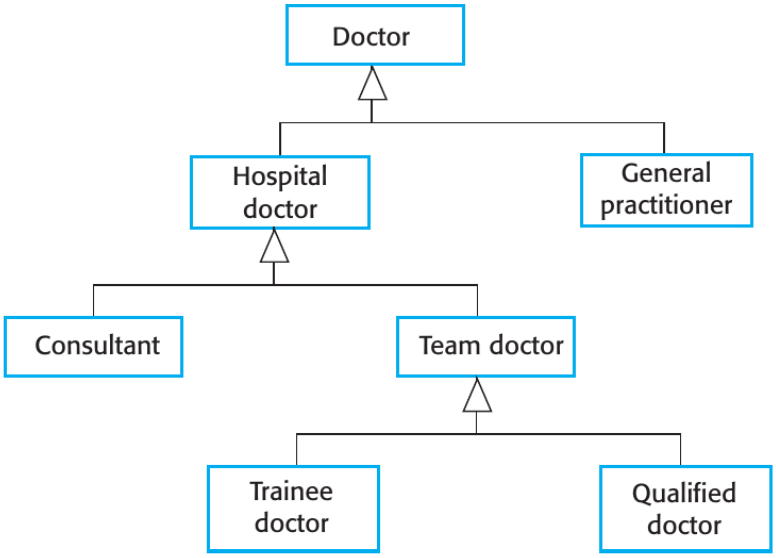

Figure 5.11 A generalization hierarchy

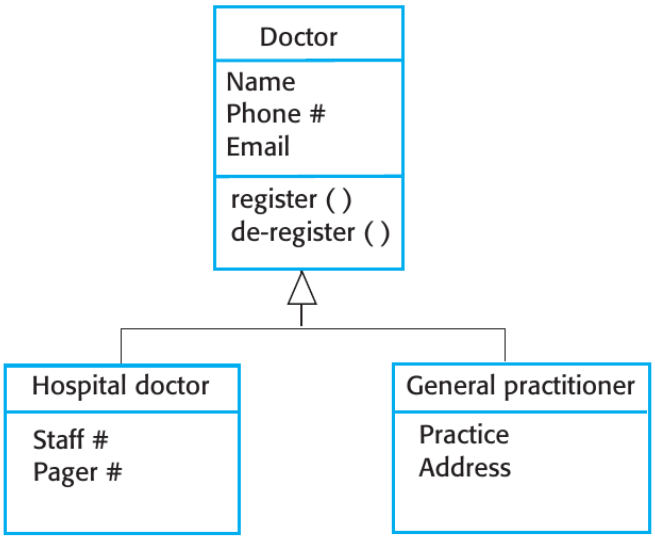

Figure 5.12 A generalization hierarchy with added detail

5.3.3 Aggregation

1. **Concept of Aggregation**:
   - Represents the part-whole relationship between objects.
   - Useful for depicting how one object is composed of several other objects.
2. **Real-world Example**:
   - A study pack for a course could include components like a book, PowerPoint slides, quizzes, and further reading recommendations.
3. **UML Representation**:
   - In UML, aggregation is a special type of association between classes.

- A diamond shape is added next to the class that represents the "whole" to indicate aggregation.

4. **Specific Example in System Modeling**:
   - A patient record could be an aggregate of the "Patient" class and an indefinite number of "Consultations."
   - This means the patient record maintains both personal patient information and individual records for each consultation with a doctor.

Aggregation is a concept in system modeling used to represent part-whole relationships between objects. It is visually represented in UML diagrams using a diamond shape next to the class that serves as the "whole." The concept is useful for showing how complex objects are composed of simpler parts.

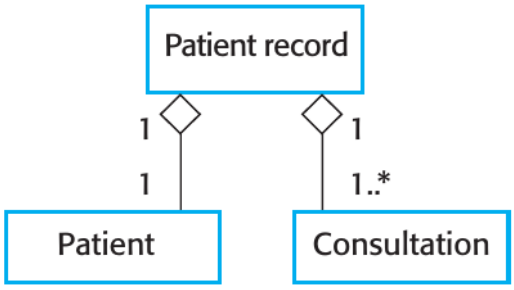

Figure 5.13 The aggregation association

5.4 Behavioral models

1. **Definition**:
   - Models that describe the dynamic behavior of a system during its execution.
   - Illustrate how a system responds to stimuli from its environment.
2. **Types of Stimuli**:
   - Stimuli triggering system behavior can be either data or events.
     a. **Data-driven Stimuli**: Availability of data triggers system processing.
     b. **Event-driven Stimuli**: Specific events trigger system processing, may or may not have associated data.
3. **Business Systems vs Real-Time Systems**:
   - **Business Systems**:
     - Primarily data-driven.
     - Controlled by data input and involve a sequence of actions on that data.
     - Example: Phone billing system calculates costs based on call data and generates a bill.
   - **Real-Time Systems**:
     - Primarily event-driven.
     - Respond to real-time events with limited data processing.
     - Example: Landline phone switching system responds to "handset activated" event by generating a dial tone.

Behavioral models are crucial for understanding how a system interacts with its environment and responds to various stimuli, which can be either data-driven or event-driven. The nature of the system—whether it's a

business system or a real-time system—determines its primary mode of operation.

5.4.1 Data-driven modelling

1. **Purpose**:
   - Illustrates the sequence of actions for processing input data to generate an output.
   - Useful for requirement analysis as it shows end-to-end processing in a system.
2. **Historical Context**:
   - One of the first graphical software models.
   - Originated in the 1970s with the use of data-flow diagrams (DFDs).
3. **Data-Flow Diagrams (DFDs)**:
   - Show how data associated with a process moves through the system.
   - Simple and intuitive, making them accessible to stakeholders including non-technical ones.
   - Can be represented in UML using activity diagrams.
4. **UML Sequence Diagrams**:
   - An alternative to DFDs for showing sequential data processing.
   - Messages are sent from left to right to show sequence.
   - Focuses on objects in the system.
5. **DFDs vs Sequence Diagrams**:
   - DFDs focus on operations or activities.
   - Sequence Diagrams focus on objects and their interactions.
   - Nonexperts often find DFDs more intuitive, while engineers may prefer sequence diagrams.

Data-driven models are instrumental in showing the sequence of actions that occur from the time an initial input is processed until an output is generated. These models can be presented in various forms such as Data-Flow Diagrams (DFDs) or UML Sequence Diagrams, each with its own set of advantages and limitations.

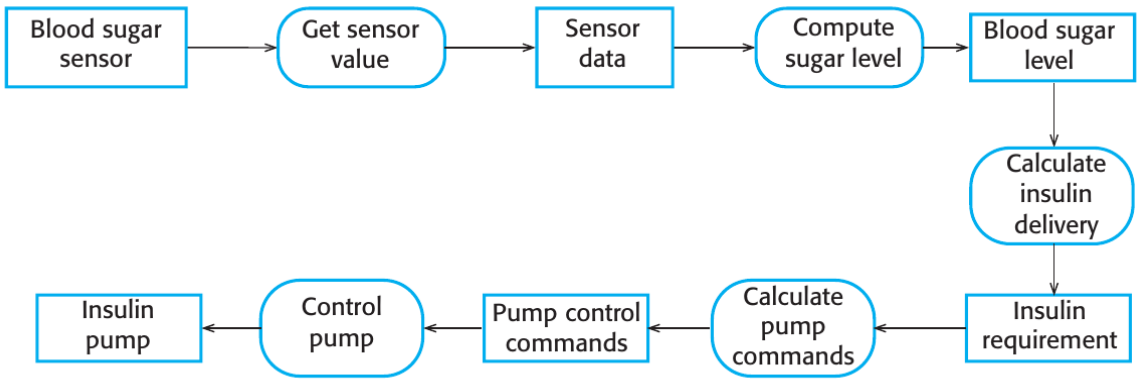

Figure 5.14 An activity model of the insulin pump's operation

processing steps in a system. Data-flow models are useful because tracking and documenting how data associated with a particular process moves through the system help analysts and designers understand what is going on in the process. DFDs are

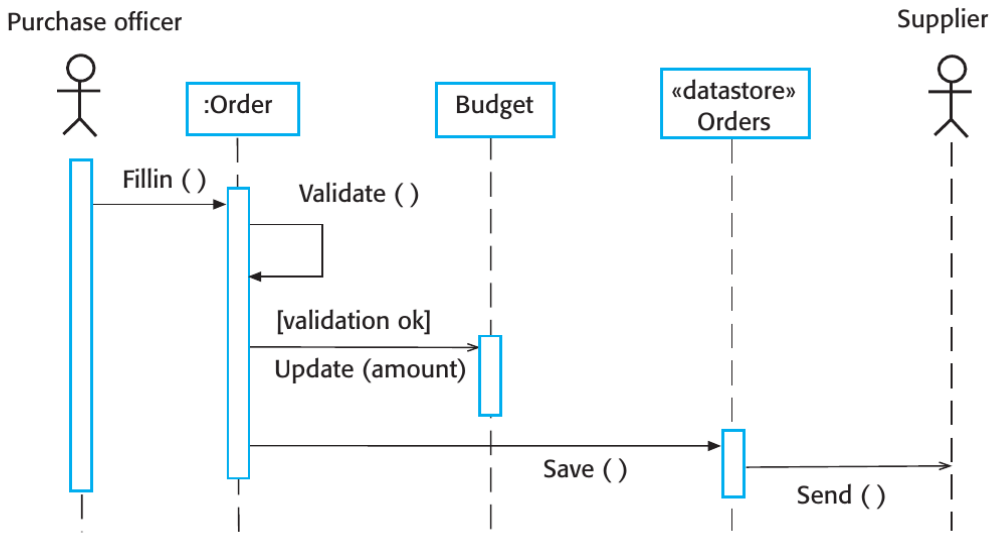

Figure 5.15 Order processing

5.4.2 Event-driven modelling

1. **Purpose**:
   - To show how a system responds to internal and external events.
   - Particularly relevant for real-time systems.
2. **Concepts**:
   - Assumes the system has a finite number of states.
   - Events (stimuli) trigger transitions between these states.
3. **UML State Diagrams**:
   - Used to represent event-based modeling.
   - Show system states and transitions but not data flow.
   - Rounded rectangles represent states and labeled arrows signify stimuli triggering transitions.
4. **Example**:
   - A simplified microwave oven model used to illustrate the approach.
   - Sequence of actions defined such as selecting power level, setting cooking time, and starting the oven.
   - Safety features like not operating when the door is open are included.
5. **Complexity Issue**:
   - Number of possible states can increase rapidly for large systems.
   - One solution is to use "superstates" to encapsulate multiple states, simplifying the high-level view.
6. **Detailed Description**:
   - State models are often extended with tables listing states, events, and descriptions for more clarity.

Event-driven modeling is a technique primarily used for real-time systems to show how a system responds to various events or stimuli. The UML supports this modeling approach through state diagrams, which depict the finite states of a system and the events that trigger transitions between these states. However, dealing with a large number of states can become complex, and thus, techniques like using "superstates" are employed to manage this complexity.

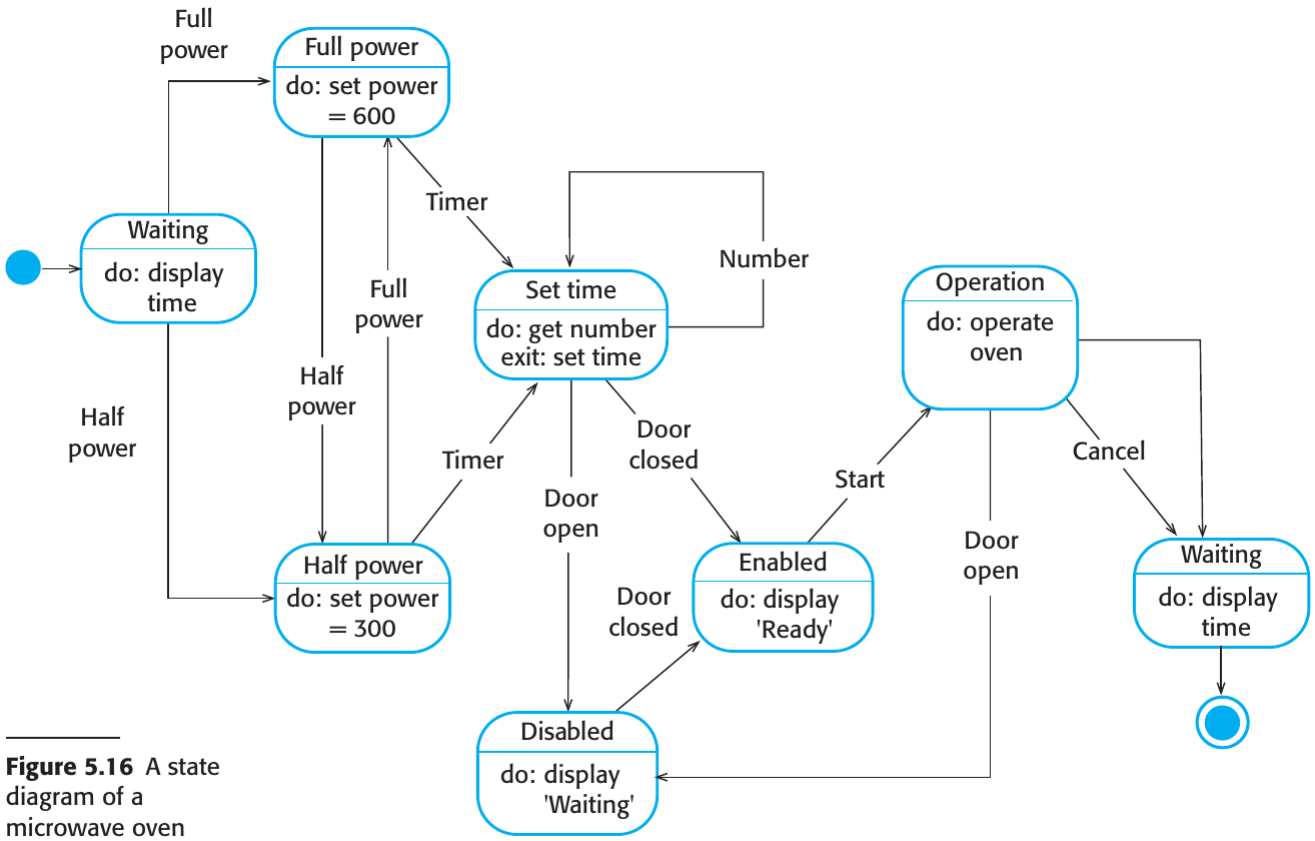

Figure 5.16 A state diagram of a microwave oven

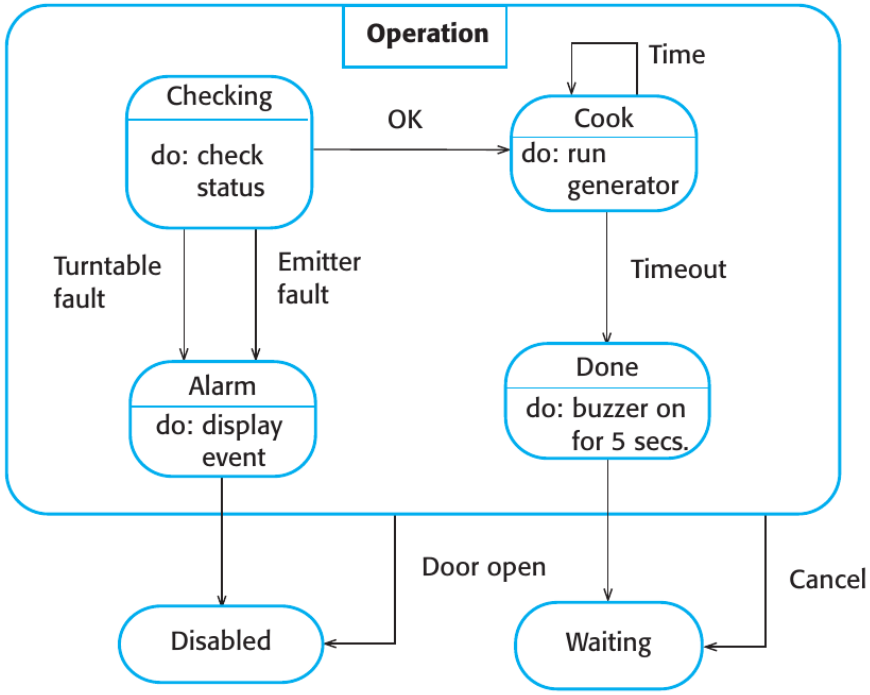

Figure 5.17 A state model of the Operation state

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows "Half power."
Full power	The oven power is set to 600 watts. The display shows "Full power."
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows "Not ready."
Enabled	Oven operation is enabled. Interior oven light is off. Display shows "Ready to cook."
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows "Cooking complete" while buzzer is sounding.
Stimulus	**Description**
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

Figure 5.18 States and stimuli for the microwave oven

5.4.3 model-driven Engineering (MDE)

1. **Definition and Approach**:
   - Model-driven engineering is a software development method where the primary outputs are models, not programs.
   - Software to run on hardware is automatically generated from these models.
   - The approach aims to abstract away the complexities of programming languages and execution platforms.
2. **Origins**:
   - MDE evolved from the concept of Model-Driven Architecture (MDA), which was proposed by the Object Management Group (OMG).
   - MDA focuses only on the design and implementation stages of software development.
3. **Scope**:
   - Unlike MDA, MDE is concerned with the entire software engineering process.
   - This includes model-based requirements engineering, model-based development processes, and model-based testing.
4. **Adoption in Industry**:
   - MDA has been adopted by several large companies for their development processes.

- However, the broader approach of MDE has not been widely adopted across the software development life cycle.

5. **Challenges**:
   - The slow adoption of MDE is a subject of discussion, with some questioning the reasons for its limited uptake in the industry.

By focusing on models rather than code, MDE aims to elevate the level of abstraction in software engineering, thereby simplifying the development process. However, its adoption has been slow compared to its predecessor, MDA.

5.5 Model-driven architecture (MDA)

1. **Definition and Methodology**:
   - MDA is a model-centric approach to software design and implementation.
   - Uses a subset of UML models at varying levels of abstraction to describe a system.
   - In theory, it can produce working programs automatically from high-level models.
2. **Types of Models**:
   - Computation Independent Model (CIM): Focuses on domain abstractions. Multiple CIMs may exist for different system views.
   - Platform Independent Model (PIM): Describes the system operations without considering implementation specifics.
   - Platform Specific Model (PSM): Transforms PIMs to suit specific platforms. May exist in layers for different platform specifics.
3. **Advantages**:
   - High-level abstraction reduces errors and speeds up design and implementation.
   - Facilitates the generation of platform-independent application models.
   - Allows for easy adaptation to new platform technologies via model translators.
4. **Challenges and Limitations**:
   - Fully automated translation from models to code is often not feasible.
   - High-level models may not always be suitable for implementation.
   - Requires human intervention for concept mapping between different CIMs.
   - Off-the-shelf tool support may be limited, requiring custom tool development.
5. **Industry Adoption and Criticism**:
   - MDA has been slow to become mainstream for several reasons:
     a. Limited applicability in addressing major issues like requirements engineering and security.
     b. Gains from using MDA are often offset by costs of tooling and introduction.
     c. Divergence from agile methodologies, which focus less on up-front modeling.
   - Success stories are mostly from companies with systems products that have long lifetimes and well-understood domains.
6. **Productivity and Reusability**:
   - MDA has shown to increase productivity and reduce maintenance costs in some cases.
   - Particularly useful in facilitating reuse, leading to significant productivity improvements.
7. **Agile Methods and MDA**:
   - There is tension between agile methods and MDA due to differing philosophies.

- Some suggest that elements of MDA can be incorporated into agile processes, but with reservations about automated code generation.

MDA focuses on creating models to abstract away the complexities of system design and implementation. While promising in theory, the approach has seen mixed results in terms of industry adoption.

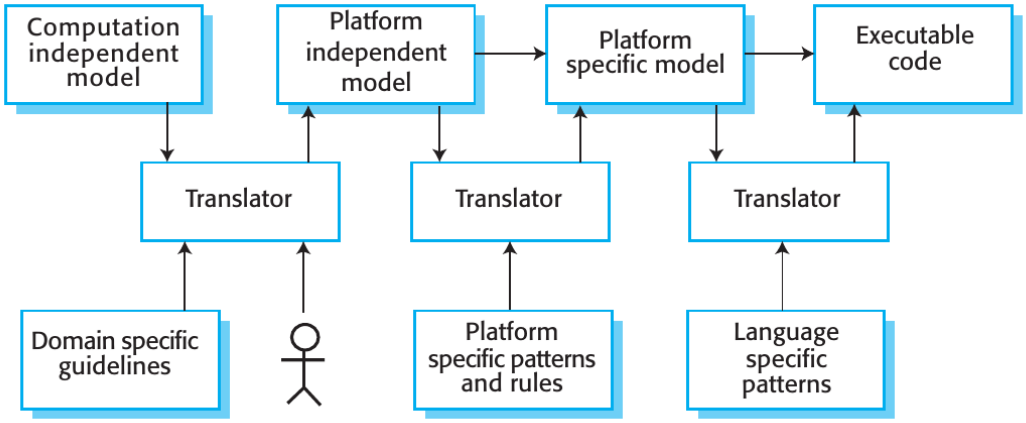

Figure 5.19 MDA transformations

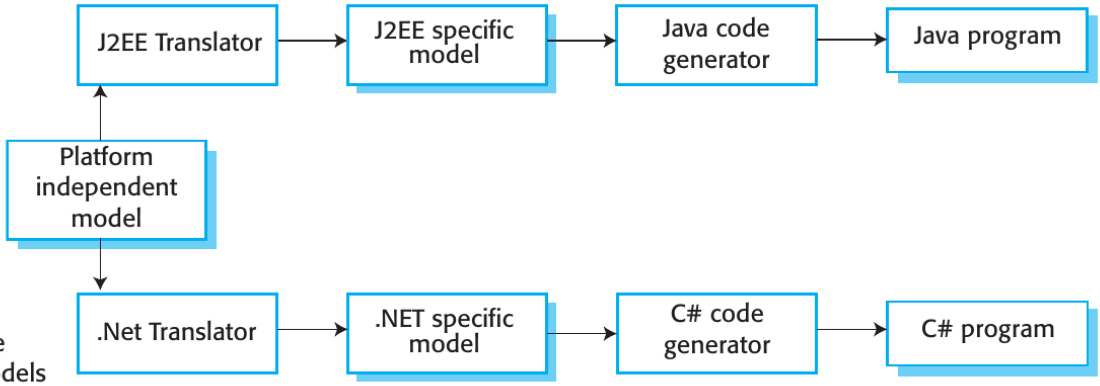

Figure 5.20 Multiple platform-specific models

Executable UML

The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible. To achieve this, you have to be able to construct graphical models with clearly defined meanings that can be compiled to executable code. You also need a way of adding information to graphical models about the ways in which the operations defined in the model are implemented. This is possible using a subset of UML 2, called Executable UML or xUML (Mellor and Balcer 2002).

http://software-engineering-book.com/web/xuml/

Summary

- A model is an abstract view of a system that deliberately ignores some system details. Complementary system models can be developed to show the system’s context, interactions, structure, and behavior.
- Context models show how a system that is being modeled is positioned in an environment with other systems and processes. They help define the boundaries of the system to be developed.
- Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
- Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.
- Behavioral models are used to describe the dynamic behavior of an executing system. This behavior can be modeled from the perspective of the data processed by the system or by the events that stimulate responses from a system.
- Activity diagrams may be used to model the processing of data, where each activity represents one process step.
- State diagrams are used to model a system’s behavior in response to internal or external events.
- Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.